# Lab 7 Instructions

## Testing Ranges by Embedding Invariant Methods

Book Pages Number: 82,83,84,85,86

There are six classes in the zip folder

1. Car.java
2. Gear.java
3. InvariantException.java
4. Moveable.java
5. SparseArray.java
6. Transmission.java

## Part 1
## Testing Ranges by Embedding Invariant Methods

The most common ranges you'll test will likely depend on data-structure concerns, not application-domain constraints.

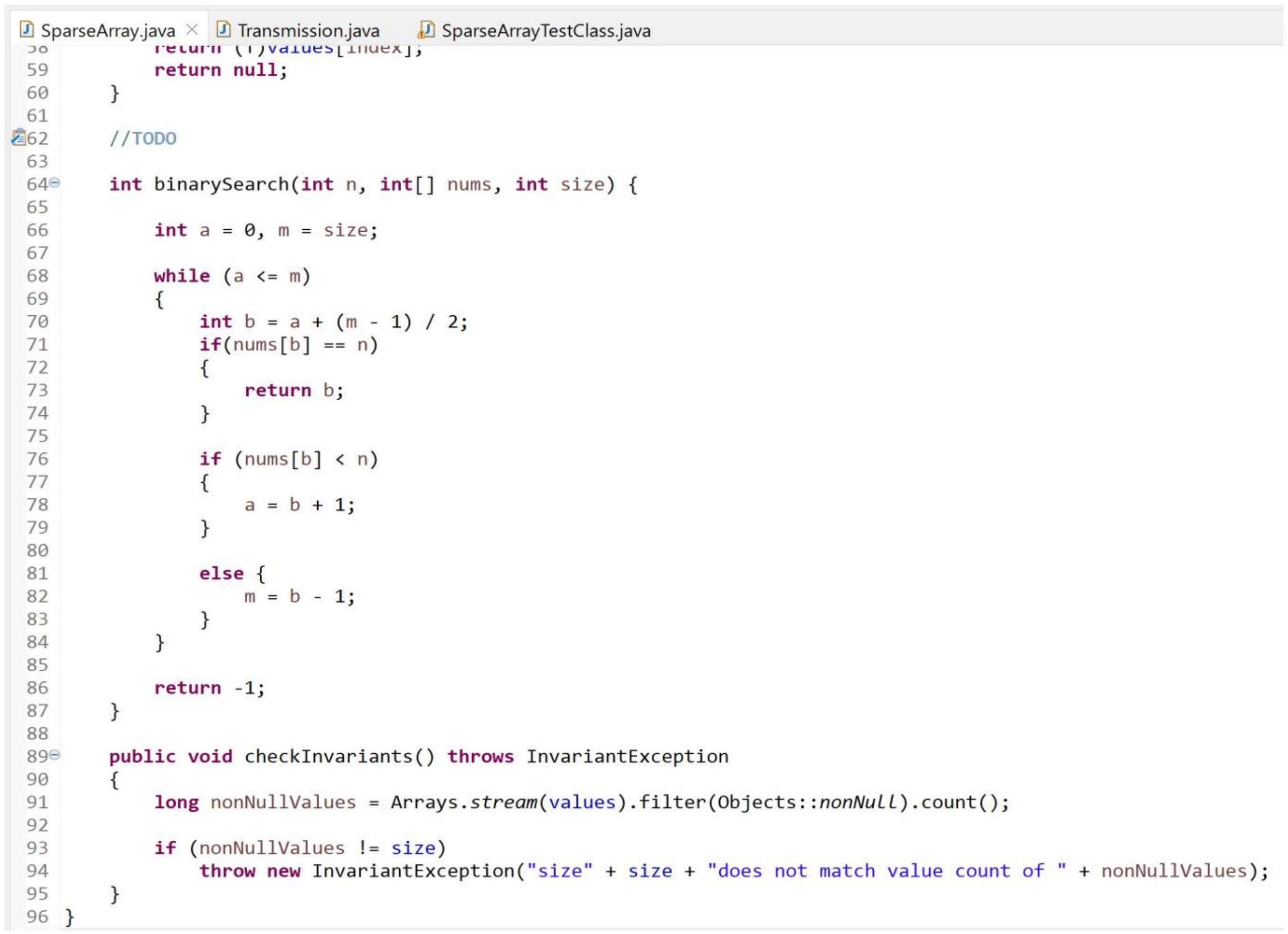
Let's look at a questionable implementation of a sparse array—a data structure designed to save space. The sweet spot for a sparse array is a broad range of indexes where most of the corresponding values are null. It accomplishes this goal by storing only non-null values, using a pair of arrays that work in concert: an array of indexes corresponds to an array of values.

1- Get the source for the "SparseArray" class from the zip file

2- TODO 1: Implement the iterative Binary search function in SparseArray.java and take screenshot of the code

//Returns index of n if it is present in nums else return -1

```java
int binarySearch(int n, int[] nums, int size)
```



```
        return (T)values[index];
        return null;
    }

    //TODO

    int binarySearch(int n, int[] nums, int size) {

        int a = 0, m = size;

        while (a <= m)
        {
            int b = a + (m - 1) / 2;
            if(nums[b] == n)
            {
                return b;
            }

            if (nums[b] < n)
            {
                a = b + 1;
            }

            else {
                m = b - 1;
            }
        }

        return -1;
    }

    public void checkInvariants() throws InvariantException
    {
        long nonNullValues = Arrays.stream(values).filter(Objects::nonNull).count();

        if (nonNullValues != size)
            throw new InvariantException("size" + size + "does not match value count of " + nonNullValues);
    }
}
```

3- **Add** the following line of code snippet to **SparseArray.java**

*public void checkInvariants() throws InvariantException*

*{*

*long nonNullValues = Arrays.stream(values).filter(Objects::nonNull).count();*

*if (nonNullValues != size)*

*throw new InvariantException("size " + size + " does not match value count of " + nonNullValues);*

*}*

4- TODO 2: Write the following new test class in SparseArrayTestClass.java. Run the code, and take screenshots of the code and test case output

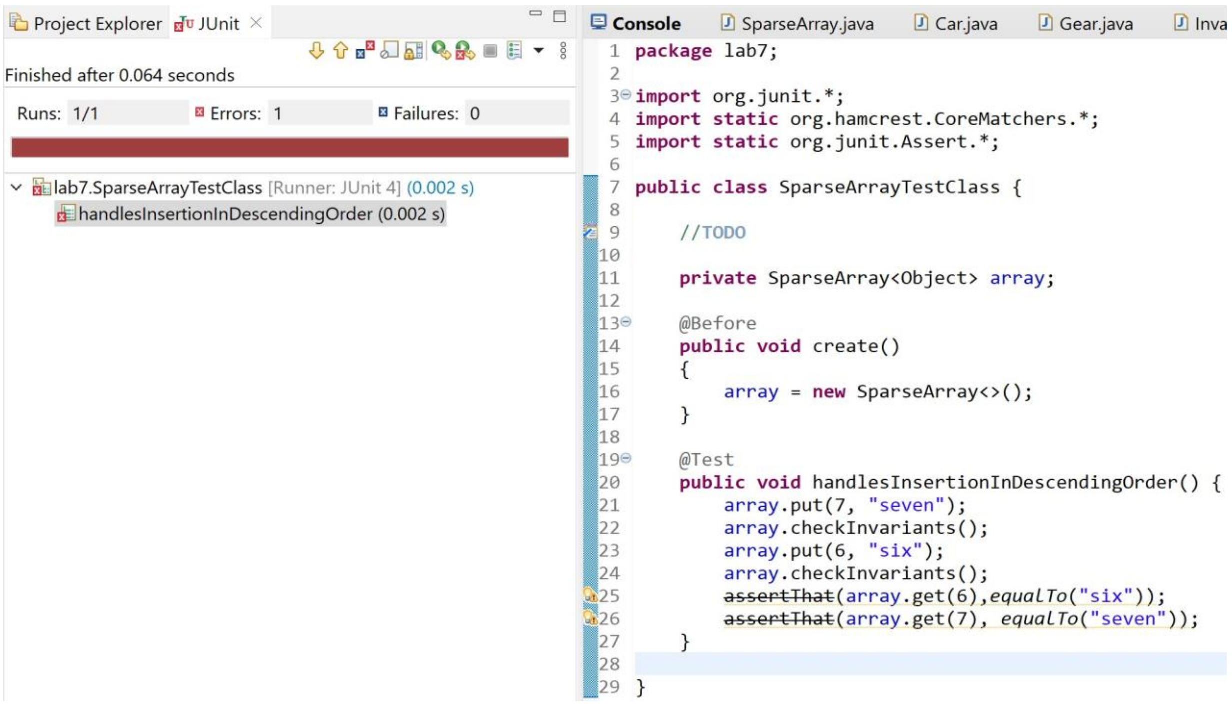
*@Test*

*public void handlesInsertionInDescendingOrder() {*

*array.put(7, "seven");*
*array.checkInvariants();*
*array.put(6, "six");*
*array.checkInvariants();*
*assertThat(array.get(6), equalTo("six"));*
*assertThat(array.get(7), equalTo("seven"));*
*}*

Add @Before annotation. public void create() method sets the new SparseArray() object. Try to create that method with @before annotation creating new object SparseArray<Object> setting it equal to array. Also, think about importing static hamcrest and Assert. (import static org.junit.Assert.*; import static org.hamcrest.CoreMatchers.*;).

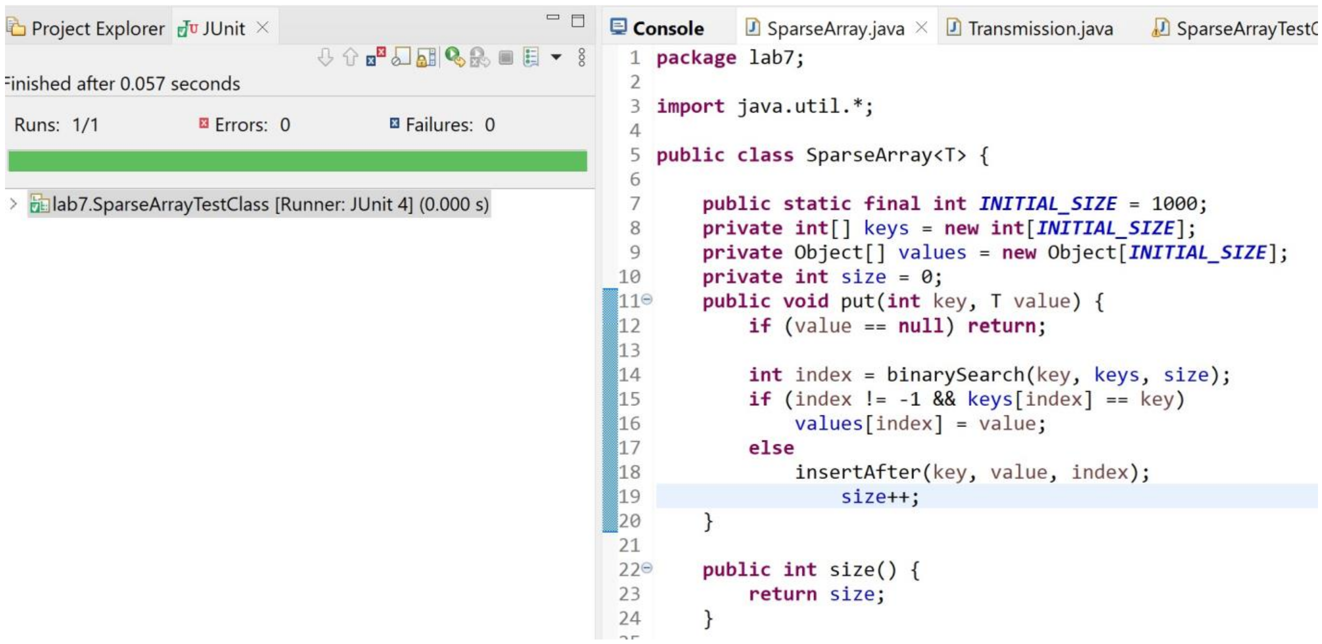
The test errors out with an InvariantException:
*util.InvariantException: size 0 does not match value count of 1 at util.SparseArray.checkInvariants(SparseArray.java:48) at util.SparseArrayTest .handlesInsertionInDescendingOrder(SparseArrayTest.java:65) …*



```
Finished after 0.064 seconds
Runs: 1/1    Errors: 1    Failures: 0

lab7.SparseArrayTestClass [Runner: JUnit 4] (0.002 s)
    handlesInsertionInDescendingOrder (0.002 s)
```

```
package lab7;

import org.junit.*;
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

public class SparseArrayTestClass {

    //TODO

    private SparseArray<Object> array;

    @Before
    public void create()
    {
        array = new SparseArray<>();
    }

    @Test
    public void handlesInsertionInDescendingOrder() {
        array.put(7, "seven");
        array.checkInvariants();
        array.put(6, "six");
        array.checkInvariants();
        ~~assertThat~~(array.get(6),equalTo("six"));
        ~~assertThat~~(array.get(7), equalTo("seven"));
    }

}
```

Take screenshots of the passed test cases and your fixed code and upload it to the lab report.Explain in your lab report what your approach to fix the code
**Hint**: Take a look at the put method in SparseArray class.

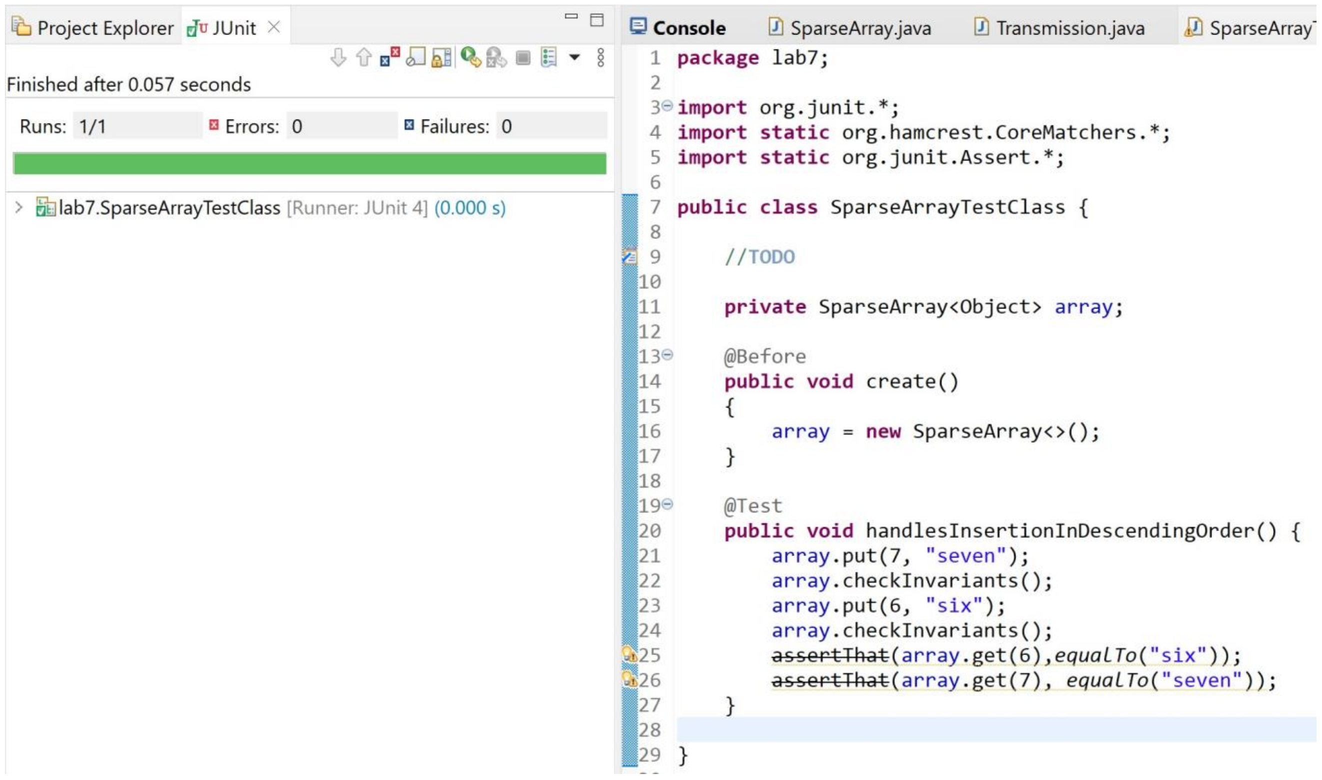


Finished after 0.057 seconds

Runs: 1/1 Errors: 0 Failures: 0

lab7.SparseArrayTestClass [Runner: JUnit 4] (0.000 s)

```java
package lab7;

import java.util.*;

public class SparseArray<T> {

    public static final int INITIAL_SIZE = 1000;
    private int[] keys = new int[INITIAL_SIZE];
    private Object[] values = new Object[INITIAL_SIZE];
    private int size = 0;
    public void put(int key, T value) {
        if (value == null) return;

        int index = binarySearch(key, keys, size);
        if (index != -1 && keys[index] == key)
            values[index] = value;
        else
            insertAfter(key, value, index);
                size++;
    }

    public int size() {
        return size;
    }
```



Finished after 0.057 seconds

Runs: 1/1 Errors: 0 Failures: 0

lab7.SparseArrayTestClass [Runner: JUnit 4] (0.000 s)

```java
package lab7;

import org.junit.*;
import static org.hamcrest.CoreMatchers.*;
import static org.junit.Assert.*;

public class SparseArrayTestClass {

    //TODO

    private SparseArray<Object> array;

    @Before
    public void create()
    {
        array = new SparseArray<>();
    }

    @Test
    public void handlesInsertionInDescendingOrder() {
        array.put(7, "seven");
        array.checkInvariants();
        array.put(6, "six");
        array.checkInvariants();
        ~~assertThat~~(array.get(6),equalTo("six"));
        ~~assertThat~~(array.get(7), equalTo("seven"));
    }

}
```

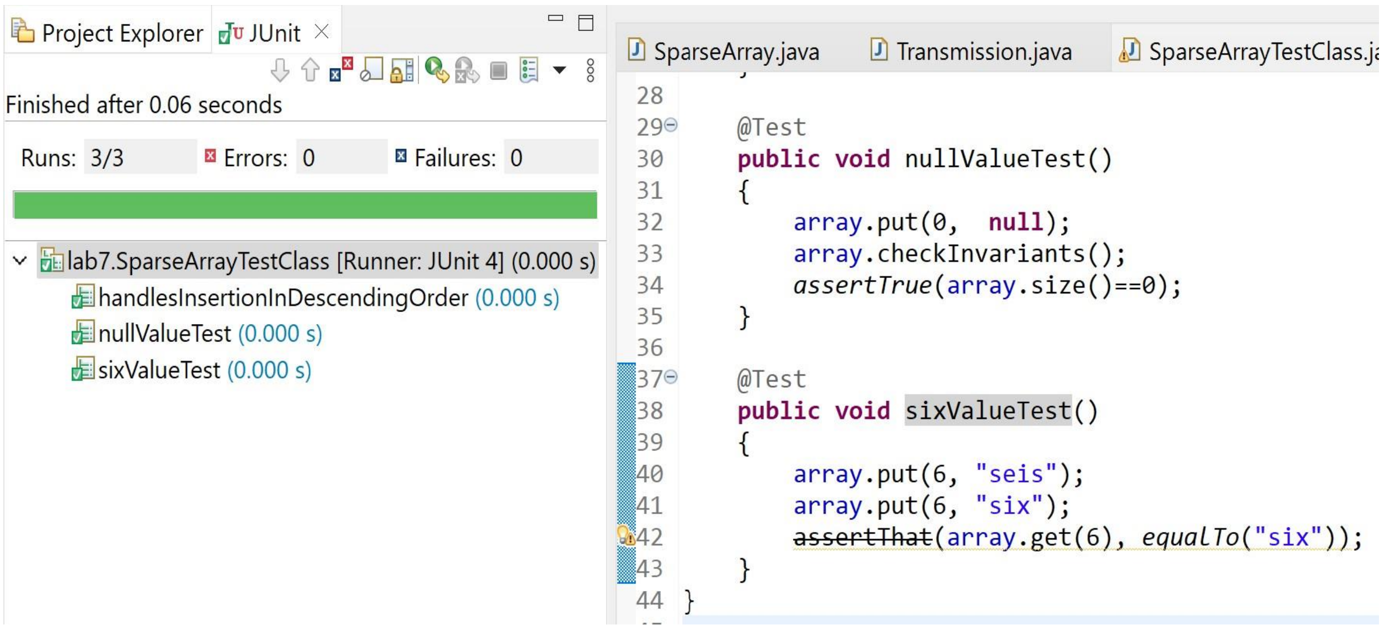
Our code indeed has a problem with tracking the internal size.
Q) What is the problem? Answer this in your lab report and fix the code in order to make the test pass.

-> The problem with the SparseArray code was that, in the ‘put’ function, the size was not being updated after the binary search function was being called. This is why the initial test case failed as the size of the array was not being updated after adding the elements. Hence, to solve the problem, “size++” was added to the put function, to allow the code to increase the size of the array. This solved the issue.

5- TODO 3: Write two test cases for the following conditions. Run the 2 test codes, and take screenshots of the code and test cases outputs

- Test case 1 (insert null value): insert (key: 0, value: null) then call checkInvariants method
  - Expected result: array size should equal to 0

- Test case 2 (insert replace value): insert (key: 6, value: “seis”) and insert again with (key: 6, value: “six”)
  - Expected result: array.get(6) should equal to “six”



```
Finished after 0.06 seconds
Runs: 3/3    Errors: 0    Failures: 0

lab7.SparseArrayTestClass [Runner: JUnit 4] (0.000 s)
    handlesInsertionInDescendingOrder (0.000 s)
    nullValueTest (0.000 s)
    sixValueTest (0.000 s)
```

```java
    @Test
    public void nullValueTest()
    {
        array.put(0,  null);
        array.checkInvariants();
        assertTrue(array.size()==0);
    }

    @Test
    public void sixValueTest()
    {
        array.put(6, "seis");
        array.put(6, "six");
        ~~assertThat~~(array.get(6), equalTo("six"));
    }
}
```

## Part 2
## Correct Reference (Correct Initial State)

When testing a method, consider:

• What it references outside its scope

• What external dependencies it has

• Whether it depends on the object being in a certain state

• Any other conditions that must exist

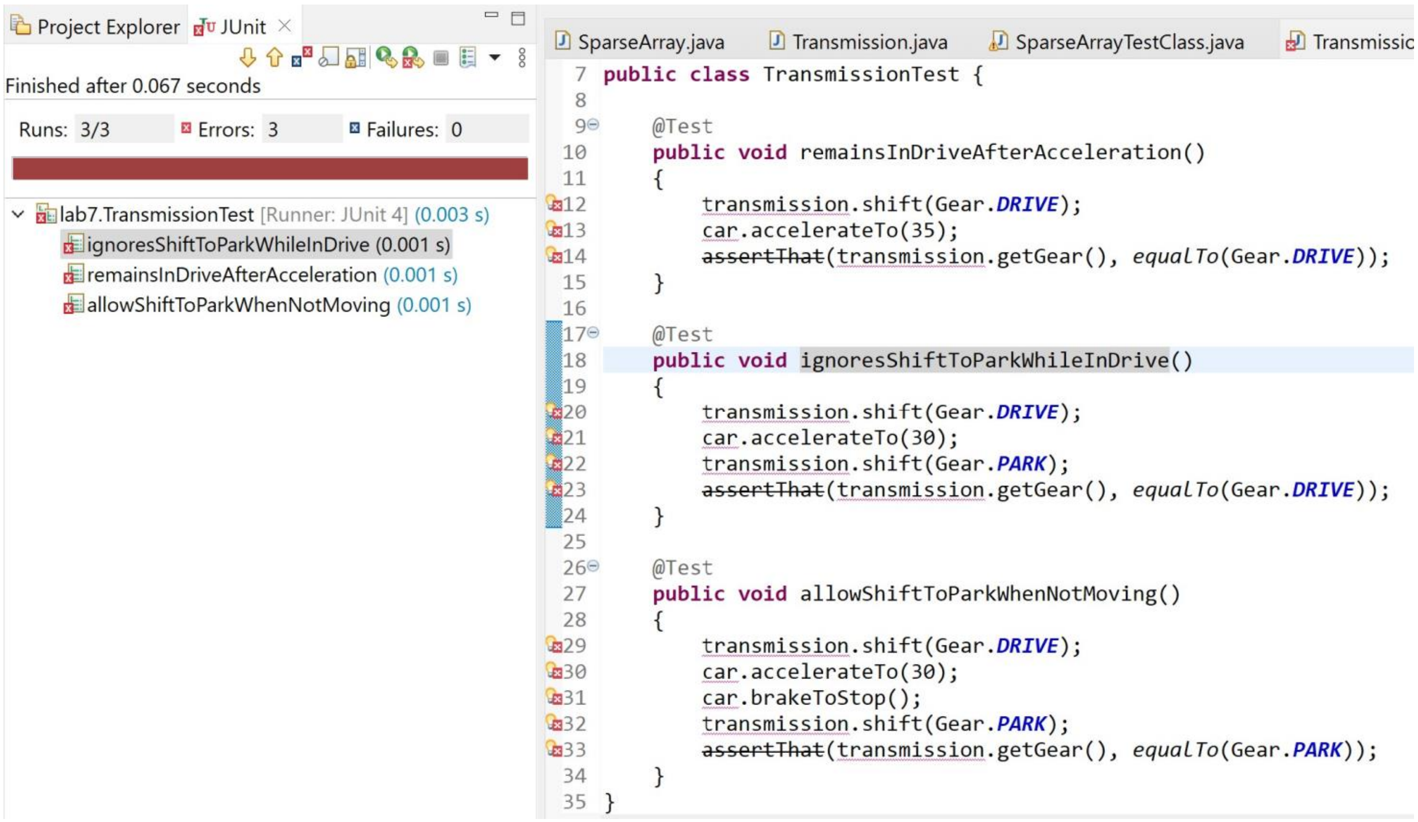
A web app that displays a customer's account history might require the cus- tomer to be logged on. The pop() method for a stack requires a nonempty stack. Shifting your car's transmission from Drive to Park requires you to first stop—if your transmission allowed the shift while the car was moving, it'd likely deliver some hefty damage to your fine Geo Metro.

When you make assumptions about any state, you should verify that your code is reasonably well-behaved when those assumptions are not met. Imagine you're developing the code for your car's microprocessor-controlled transmission. You want tests that demonstrate how the transmission behaves when the car is moving versus when it is not. Our tests for the Transmission code cover three critical scenarios: that it remains in Drive after accelerating, that it ignores the damaging shift to Park while in Drive, and that it *does* allow the shift to Park once the car isn't moving.

6- TODO 4: Inspect and run the following TransmissionTest code. Take a screenshot of the code and output

```
@Test

public void remainsInDriveAfterAcceleration() {

transmission.shift(Gear.DRIVE);
 car.accelerateTo(35);
assertThat(transmission.getGear(),equalTo(Gear.DRIVE));

}

@Test
public void ignoresShiftToParkWhileInDrive() {
transmission.shift(Gear.DRIVE);
car.accelerateTo(30);
transmission.shift(Gear.PARK);
assertThat(transmission.getGear(),equalTo(Gear.DRIVE)); }
```

```
@Test
 public void allowsShiftToParkWhenNotMoving()
{
 transmission.shift(Gear.DRIVE);
 car.accelerateTo(30);
  car.brakeToStop();
 transmission.shift(Gear.PARK);
 assertThat(transmission.getGear(), equalTo(Gear.PARK));
}
```


Project Explorer | JUnit

Finished after 0.067 seconds

Runs: 3/3 Errors: 3 Failures: 0

```
lab7.TransmissionTest [Runner: JUnit 4] (0.003 s)
    ignoresShiftToParkWhileInDrive (0.001 s)
    remainsInDriveAfterAcceleration (0.001 s)
    allowShiftToParkWhenNotMoving (0.001 s)
```

SparseArray.java | Transmission.java | SparseArrayTestClass.java | Transmissic

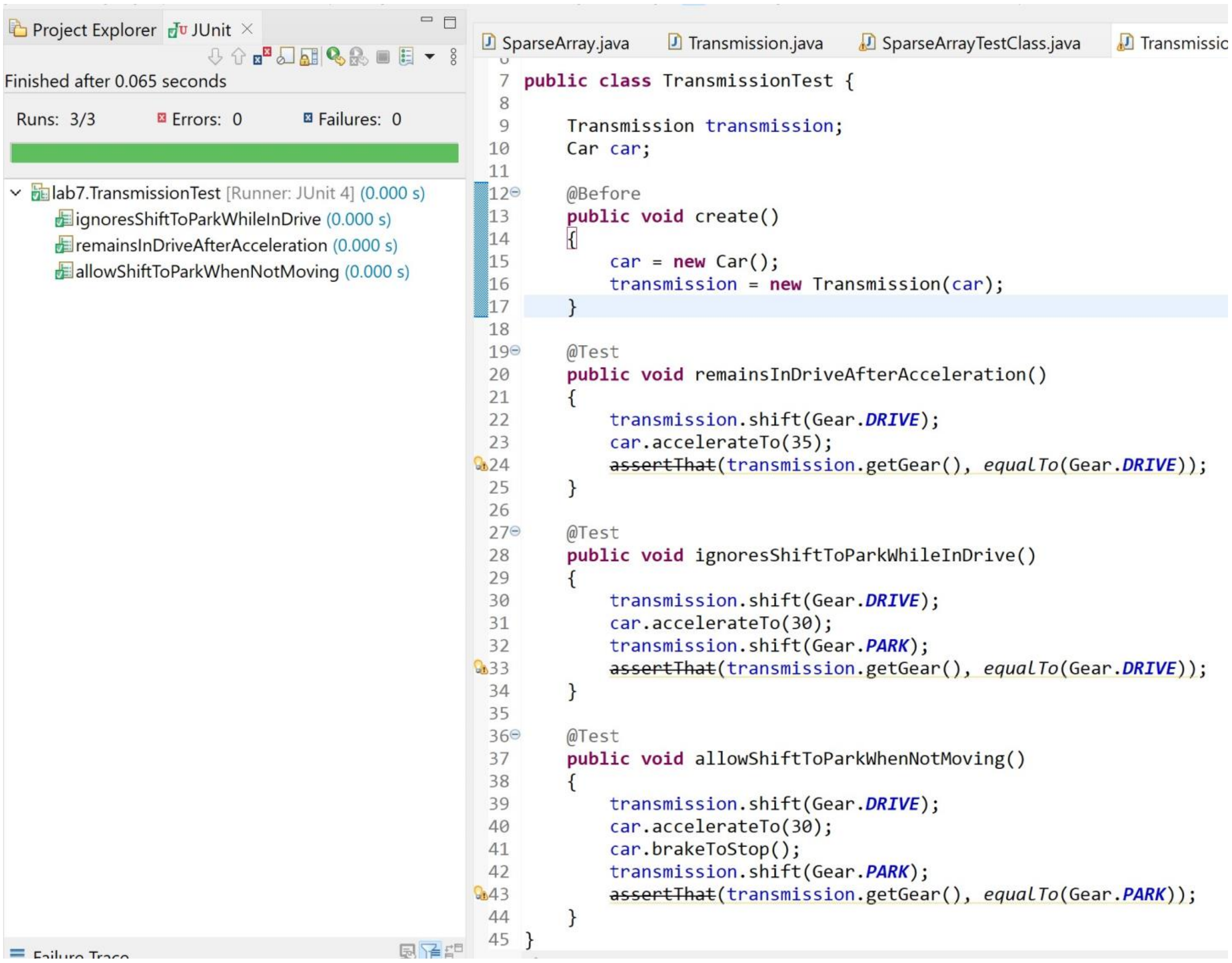
```
public class TransmissionTest {

    @Test
    public void remainsInDriveAfterAcceleration()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(35);
        ~~assertThat~~(transmission.getGear(), equalTo(Gear.DRIVE));
    }

    @Test
    public void ignoresShiftToParkWhileInDrive()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(30);
        transmission.shift(Gear.PARK);
        ~~assertThat~~(transmission.getGear(), equalTo(Gear.DRIVE));
    }

    @Test
    public void allowShiftToParkWhenNotMoving()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(30);
        car.brakeToStop();
        transmission.shift(Gear.PARK);
        ~~assertThat~~(transmission.getGear(), equalTo(Gear.PARK));
    }
}
```

The *preconditions* for a method represent the state things must be in for it to run. The precondition for putting a transmission in Park is that the car must be at a standstill. We want to ensure that the method behaves gracefully when its precondition isn't met (in our case, we ignore the Park request).

*Postconditions* state the conditions that you expect the code to make true— essentially, the assert portion of your test. Sometimes this is simply the return value of a called method. You might also need to verify other *side effects*—changes to state that occur as a result of invoking behavior.

In the allowsShiftToParkWhenNotMoving test case, calling brakeToStop() on the car instance has the side effect of setting the car's speed to 0

7- - TODO 5: The code above will not run. You will need to fix the test code as follows and take a screenshot:
Add @before annotation and think about public void create() method.
You are creating new Car object and a new transmission where newly created car object should pass in it.
Also, think about importing static hamcrest and Assert.



```
Finished after 0.065 seconds

Runs: 3/3     Errors: 0     Failures: 0

lab7.TransmissionTest [Runner: JUnit 4] (0.000 s)
    ignoresShiftToParkWhileInDrive (0.000 s)
    remainsInDriveAfterAcceleration (0.000 s)
    allowShiftToParkWhenNotMoving (0.000 s)
```

```java
public class TransmissionTest {

    Transmission transmission;
    Car car;

    @Before
    public void create()
    {
        car = new Car();
        transmission = new Transmission(car);
    }

    @Test
    public void remainsInDriveAfterAcceleration()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(35);
        assertThat(transmission.getGear(), equalTo(Gear.DRIVE));
    }

    @Test
    public void ignoresShiftToParkWhileInDrive()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(30);
        transmission.shift(Gear.PARK);
        assertThat(transmission.getGear(), equalTo(Gear.DRIVE));
    }

    @Test
    public void allowShiftToParkWhenNotMoving()
    {
        transmission.shift(Gear.DRIVE);
        car.accelerateTo(30);
        car.brakeToStop();
        transmission.shift(Gear.PARK);
        assertThat(transmission.getGear(), equalTo(Gear.PARK));
    }
}
```

## Part 3
## Submission

**Upload the following:**

1. Screenshots of test passing and the answers to the mentioned questions above in the different parts.

2. Answer to “what does checkvariants() method do?”

→ The checkvariants() method tests that if there are any variants that fail to hold the value “true”, it will throw an exception. It also tests the probe at the value that is being stored in the array, to see if an exception needs to be thrown or not.

3. Answer to “what Transmission.Java class does?”

➔ The Transmission.Java class checks the current state of the car, such as if it is moving or not. It then allows the user to shift the gear to either DRIVE or PARK, depending on the state of the car. For example, if the state of the car is “Moving”, it will not allow the user to shift the gear to PARK, until it stops.